MultiMarkdown v6 Quick Start Guide

Fletcher T. Penney

March 10, 2017

Contents

Introduction

Version: 6.0-b

This document serves as a description of MultiMarkdown (MMD) v6, as well as a sample document to demonstrate the various features. Specifically, differences from MMD v5 will be pointed out.

Performance

A big motivating factor leading to the development of MMD v6 was performance. When MMD first migrated from Perl to C (based on peg- markdown[1]), it was among the fastest Markdown parsers available. That was many years ago, and the "competition" has made a great deal of progress since that time.

[1] https://github.com/jgm/peg-markdown

When developing MMD v6, one of my goals was to keep MMD at least in the ballpark of the fastest processors. Of course, being *the* fastest would be fantastic, but I was more concerned with ensuring

that the code was easily understood, and easily updated with new features in the future.

MMD v3 – v5 used a PEG to handle the parsing. This made it easy to understand the relationship between the MMD grammar and the parsing code, since they were one and the same. However, the parsing code generated by the parsers was not particularly fast, and was prone to troublesome edge cases with terrible performance characteristics.

The first step in MMD v6 parsing is to break the source text into a series of tokens, which may consist of plain text, whitespace, or special characters such as '*', '[', etc. This chain of tokens is then used to perform the actual parsing.

MMD v6 divides the parsing into two separate phases, which actually fits more with Markdown's design philosophically.

1. Block parsing consists of identifying the "type" of each line of the source text, and grouping the lines into blocks (e.g. paragraphs, lists, blockquotes, etc.) Some blocks are a single line (e.g. ATX headers), and others can be many lines long. The block parsing in MMD v6 is handled by a parser generated by lemon[2]. This parser allows the block structure to be more readily understood by non-programmers, but the generated parser is still fast.

2. Span parsing consists of identifying Markdown/MMD structures that occur inside of blocks, such as links, images, strong, emph, etc. Most of these structures require matching pairs of tokens to specify where the span starts and where it ends. Most of these spans allow arbitrary levels of nesting as well. This made parsing them correctly in the PEG-based code difficult and slow. MMD v6 uses a different approach that is accurate and has good performance characteristics even with edge cases. Basically, it keeps a stack of each "opening" token as it steps through the token chain. When a "closing" token is found, it is paired with the most recent appropriate opener on the stack. Any tokens in between the opener and closer are removed, as they are not able to be matched any more. To avoid unnecessary searches for non- existent openers, the parser keeps track of which opening tokens have been discovered. This allows the parser to continue moving forwards without having to go backwards and re-parse any previously visited tokens.

[2] http://www.hwaci.com/sw/lemon/

The result of this redesigned MMD parser is that it can parse short documents more quickly than CommonMark[3], and takes only 15% – 20% longer to parse long documents. I have not delved too deeply into this, but I presume that CommonMark has a bit more "set-up"

[3] http://commonmark.org/

time that becomes expensive when parsing a short document (e.g. a paragraph or two). But this cost becomes negligible when parsing longer documents (e.g. file sizes of 1 MB). So depending on your use case, CommonMark may well be faster than MMD, but we're talking about splitting hairs here.... Recent comparisons show MMD v6 taking approximately 4.37 seconds to parse a 108 MB file (approximately 24.8 MB/second), and CommonMark took 3.72 seconds for the same file (29.2 MB/second). For comparison, MMD v5.4 took approximately 94 second for the same file (1.15 MB/second).

For a more realistic file of approx 28 kb (the source of the Markdown Syntax web page), both MMD and CommonMark parse it too quickly to accurately measure. In fact, it requires a file consisting of the original file copied 32 times over (0.85 MB) before `/usr/bin/env time` reports a time over the minimum threshold of 0.01 seconds for either program.

There is still potentially room for additional optimization in MMD. However, even if I can't close the performance gap with CommonMark on longer files, the additional features of MMD compared with Markdown in addition to the increased legibility of the source code of MMD (in my biased opinion anyway) make this project worthwhile.

Parse Tree

MMD v6 performs its parsing in the following steps:

1. Start with a null-terminated string of source text (C style string)
2. Lex string into token chain
3. Parse token chain into blocks
4. Parse tokens within each block into span level structures (e.g. strong, emph, etc.)
5. Export the token tree into the desired output format (e.g. HTML, LaTeX, etc.) and return the resulting C style string

   OR

6. Use the resulting token tree for your own purposes.

The token tree (AST) includes starting offsets and length of each token, allowing you to use MMD as part of a syntax highlighter. MMD v5 did not have this functionality in the public version, in part because the PEG parsers used did not provide reliable offset positions, requiring a great deal of effort when I adapted MMD for use in MultiMarkdown Composer[4].

[4] http://multimarkdown.com/

These steps are managed using the `mmd_engine` "object". An individual `mmd_engine` cannot be used by multiple threads simultaneously, so if libMultiMarkdown is to be used in a multithreaded program, a separate `mmd_engine` should be created for each thread. Alternatively, just use the slightly more abstracted `mmd_convert_string()` function that handles creating and destroying the `mmd_engine` automatically.

Features

Abbreviations (Or Acronyms)

This file includes the use of MMD as an abbreviation for MultiMarkdown. The abbreviation will be expanded on the first use, and the shortened form will be used on subsequent occurrences.

Abbreviations can be specified using inline or reference syntax. The inline variant requires that the abbreviation be wrapped in parentheses and immediately follows the >.

```
[>MMD] is an abbreviation.  So is [>(MD) Markdown].

[>MMD]: MultiMarkdown
```

There is also a "shortcut" method for abbreviations that is similar to the approach used in prior versions of MMD. You specify the definition for the abbreviation in the usual manner, but MMD will automatically identify each instance where the abbreviation is used and substitute it automatically. In this case, the abbreviation is limited to a more basic character set which includes letters, numbers, periods, and hyphens, but not much else. For more complex abbreviations, you must explicitly mark uses of the abbreviation.

Citations

Citations can be specified using an inline syntax, just like inline footnotes.

CriticMarkup

MMD v6 has improved support for CriticMarkup[5], both in terms of parsing, and in terms of support for each output format. You can insert text, ~~delete text~~, substitute ~~one thing~~for another, highlight text, and in the text.

leave comments

[5] http://criticmarkup.com/

Emph and Strong

The basics of emphasis and strong emphasis are unchanged, but the parsing engine has been improved to be more accurate, particularly in various edge cases where proper parsing can be difficult.

Fenced Code Blocks

Fenced code blocks are fundamentally the same as MMD v5, except:

1. The leading and trailing fences can be 3, 4, or 5 backticks in length. That should be sufficient to account for complex documents without requiring a more complex parser.
2. If there is no trailing fence, then everything after the leading fence is considered to be part of the code block.

Glossary Terms

If there are terms in your document you wish to define in a glossary at the end of your document, you can define them using the glossary syntax.

Glossary terms can be specified using inline or reference syntax. The inline variant requires that the abbreviation be wrapped in parentheses and immediately follows the `?`.

```
[?(glossary) The glossary collects information about important
terms used in your document] is a glossary term.

[?glossary] is also a glossary term.

[?glossary]: The glossary collects information about important
terms used in your document
```

Much like abbreviations, there is also a "shortcut" method that is similar to the approach used in prior versions of MMD. You specify the definition for the glossary term in the usual manner, but MMD will automatically identify each instance where the term is used and substitute it automatically. In this case, the term is limited to a more basic character set which includes letters, numbers, periods, and hyphens, but not much else. For more complex glossary terms, you must explicitly mark uses of the term.

Internationalization

MMD v6 includes support for substituting certain text phrases in other languages. This only affects the HTML format.

Metadata

Metadata in MMD v6 includes new support for LaTeX – the `latex config` key allows you to automatically setup of multiple `latex include` files at once. The default setups that I use would typically consist of one LaTeX file to be included at the top of the file, one to be included right at the beginning of the document, and one to be included at the end of the document. If you want to specify the latex files separately, you can use `latex leader`, `latex begin`, and `latex footer`.

Table of Contents

By placing `{{TOC}}` in your document, you can insert an automatically generated Table of Contents in your document. As of MMD v6, the native Table of Contents functionality is used when exporting to LaTeX or OpenDocument formats.

Future Steps

Some features I plan to implement at some point:

1. MMD v5 used to automatically identify abbreviated terms throughout the document and substitute them automatically. I plan to reimplement this functionality, but will probably improve upon it to include glossary terms, and possibly even support for indexing documents in LaTeX (and possibly OpenOffice).

2. OPML export support is not available in v6. I plan on adding improved support for this at some point. I was hoping to be able to re-use the existing v6 parser but it might be simpler to use the approach from v5 and earlier, which was to have a separate parser tuned to only identify headers and "stuff between headers".

3. Improved EPUB support. Currently, EPUB support is provided by a separate tool[6]. At some point, I would like to better integrate this into MMD itself.

[6] https://github.com/fletcher/MMD-ePub

Glossary

AST Abstract Syntax Tree https://en.wikipedia.org/wiki/Abstract_syntax_tree 3

glossary The glossary collects information about important terms used in your document. 4

PEG Parsing Expression Grammar https://en.wikipedia.org/wiki/Parsing_expression_grammar 1–3

Abbreviations

MMD MultiMarkdown. 1–6